27 | Kafka（一）：消息队列的新标准

2021-12-06 徐文浩

《大数据经典论文解读》 课程介绍 >

讲述：徐文浩
时长 20:43 大小 18.99M

你好，我是徐文浩。

过去的两节课里，我给你介绍了 S4 和 Storm 这两个流式计算框架相关的论文。不过，在讲解这两篇论文的时候，我们其实没有去搞清楚对应的流式数据是从哪里来的。虽然 S4 里有 Keyless PE，Storm 里也有 Spout，它们都是框架自己提供的发送流式数据的机制，这些框架本身并不能产生数据。我们各种应用服务器产生的数据，必须要想一个办法，能够给这些流式数据处理系统。

其实，不只是流式数据处理系统有这个需求，我们之前讲解过的 GFS/MapReduce 这套分布式文件系统，以及大数据批处理系统，一样面临这个“**数据从哪里来**”的问题。

这个问题，也就是我们今天要探讨的主题，就是我们应该通过一个什么样的系统，来传输数据。这个系统需要满足哪些需求，整个系统架构应该怎么设计。而对于这个问题的解答，就是开源的 Kafka 系统。

同样在 2011 年，来自 LinkedIn 的三位工程师，一起发表了《Kafka: a Distributed Messaging System for Log Processing》这样一篇论文，并且把论文里描述的这个系统 Kafka 开源。这篇论文，可以说帮我们圆上了整个大数据系统的最后一个环节，就是**高性能、高可用的数据传输**。

虽然后续，整个大数据领域仍然有不断的迭代更新，但是有了 Kafka 之后，通过 Hadoop/Spark 进行批数据处理，通过 Hive 搭建数据仓库，通过 Storm 进行流式数据处理，然后通过 Kafka 作为业务系统和大数据系统之间的消息管道，已经是一个完整而成熟的“标准方案”了。可以说，随着 Kafka 的发布，整个大数据领域开始迈入一个成熟的阶段。大部分公司都可以通过组合开源框架，搭建起完善的大数据系统，而不再需要自己去“造轮子”了。

那么，在学完今天这节课之后，希望你能够掌握以下两点：

首先是整个大数据系统的整体框架是怎么样的，为什么我们需要 Kafka 这样一个桥梁，连接起我们的应用系统、大数据批处理系统，以及大数据流式处理系统。

其次是 Kafka 系统的设计，和传统的“消息队列”系统有什么不同，以及为什么 Kafka 需要这样来设计。

大数据系统的基本框架

之前，我们所有讲解的大数据论文，都是在看大数据系统的内部设计。那么，我们这些大数据系统的**数据是从哪里来的呢？**事实上，这些大数据系统处理的，都是对应的应用系统或者业务系统产生的“日志”。最早用上这些大数据技术的，往往都是互联网公司的广告和搜索业务。你每浏览一次网页，点击一次广告，进行一次搜索，它们都会在服务器上记录下对应的访问日志。

在一开始，对于这些数据处理的需求还不是流式处理，而是通过 MapReduce 来进行处理的。所以，我们就有了一个最简单的需求，就是把每台应用服务器上的日志，放到 Hadoop 集群的 HDFS 上去。

从 cronjob 出发

当然，我们有一些简单的笨办法就能做到这一点。我们可以直接把日志落地到服务器所在的本地硬盘上，然后按照时间定时分割出一个新文件。比如，每小时服务器上就会生成一个日志文件，然后我们再通过像 Linux 下的 cronjob 这样的定时任务，把这个文件上传到 HDFS 上就好了。

这个方式非常简单粗暴，但是对应的问题也显而易见，那就是在日志上传到 HDFS 之前，整个系统是**没有灾备**的。如果我们某一台服务器出现了硬件故障，那么我们会有一段时间的日志文件没有上传到 HDFS 上，也就是日志丢了。

那么，**我们可不可以直接通过 HDFS 提供的客户端，把日志文件往 HDFS 上写呢？**这样，日志一旦写入到 HDFS 上，我们就有三份数据副本，也就不会有丢数据的问题了。

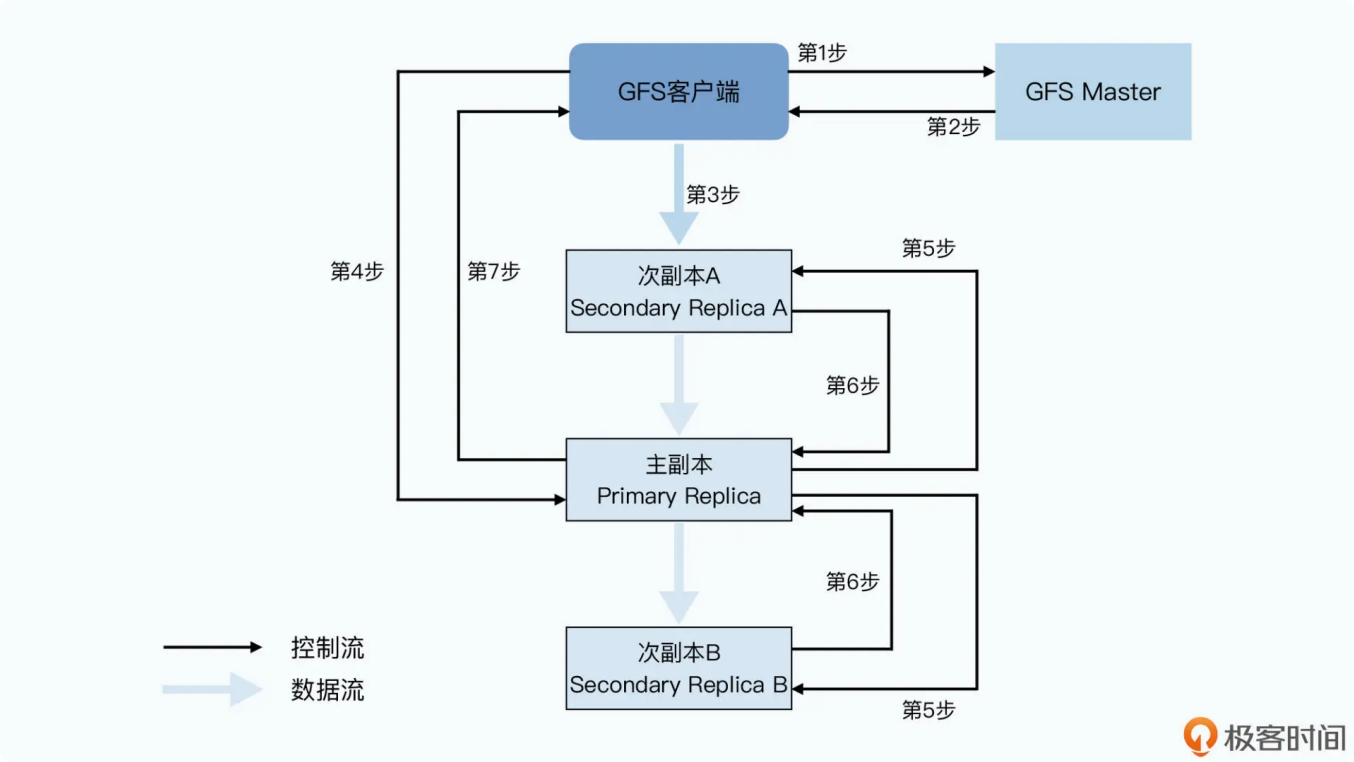
但是，这样一来，HDFS 的并发压力就会很大。如果我们有 100 台应用服务器，那么我们就会有 100 个客户端在往 HDFS 上写入日志。并且，这个写入是 7 天 24 小时无休的。

而且，你还要考虑这样一个问题，那就是你是让所有的应用服务器，各自写各自的日志文件，还是大家都往同一个日志文件里写呢？

如果是各自写各自的日志文件，而我们为了后续的 MapReduce 能够按小时去处理数据，那么我们 100 台应用服务器不管有多少日志，每小时就会至少产生 100 个文件。而大量的小文件，本身是不适合 MapReduce 这样，擅长顺序读而不是随机读的数据处理方式的。

一方面，在 HDFS 上，每个文件无论大小，至少都要占用一个 Block 也就是 64MB 大小，那么大量的小文件，就会浪费很多存储空间。另一方面，MapReduce 里，对于每个单独的文件都需要一个独立的 Map Task 来读取，这会使得我们后续处理数据的额外开销（overhead）变得很大。

而如果我们让很多个应用服务器，往同一个 HDFS 的文件里写，其实一样会遇到**性能问题**。虽然，在 HDFS（GFS）里，数据的追加写入是有一致性保障的。但是，我在前面讲 GFS 的论文的时候说过，我们往同一个文件里写，客户端之间一样会互相竞争。这个竞争发生在 chunkserver 的主副本上，所有的数据追加写入，都会在这里排队。由 chunkserver 的主副本，来决定下一条记录是由哪一个客户端来写。



GFS中的数据写入流程，如果很多个客户端都尝试在同一个文件上追加写入
那么它们会在这个文件的主副本所在的chunkserver解决冲突问题，多个客户端会在这里“排队”
也就是图里的第4步

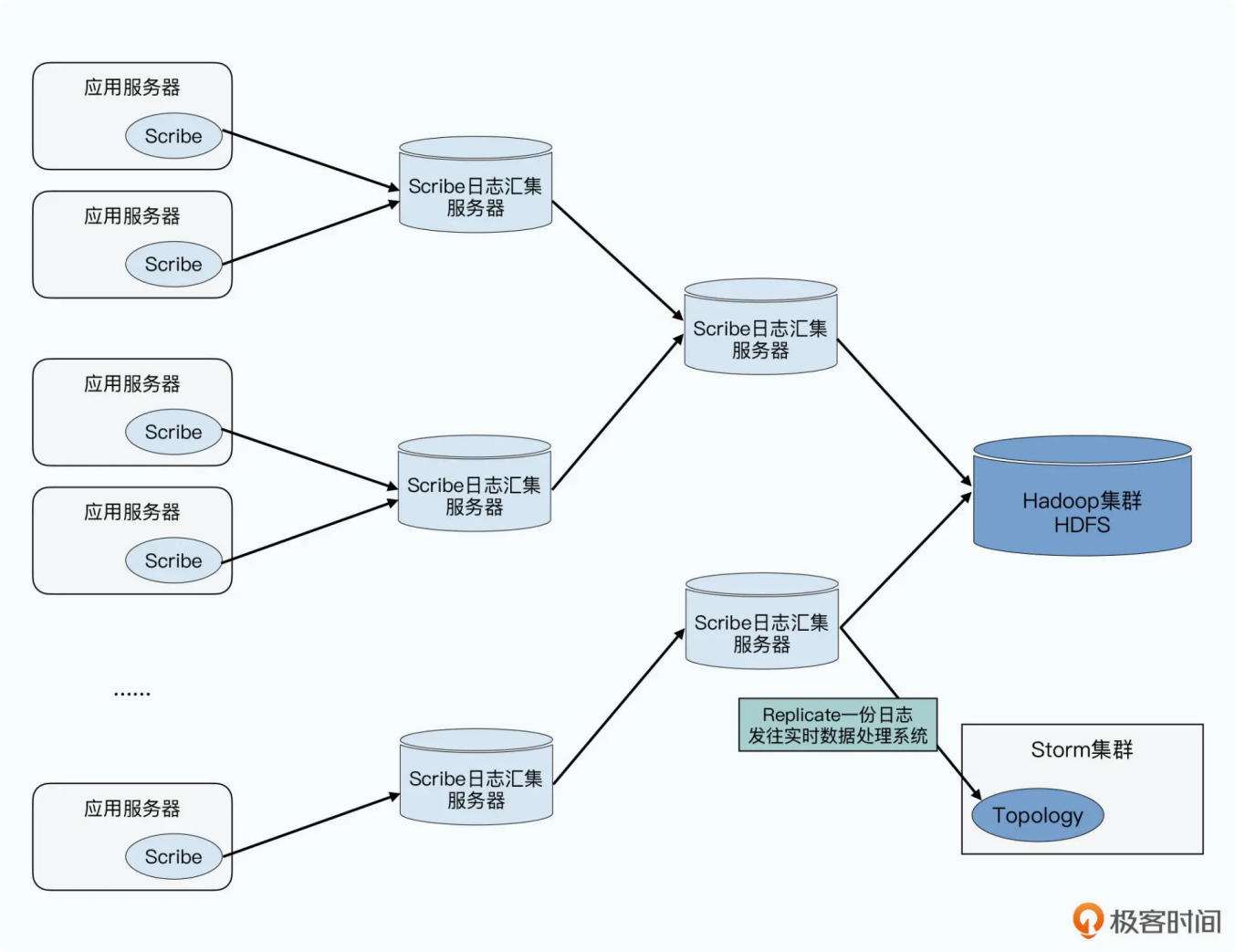
归根到底，是 HDFS（GFS）这样的分布式文件系统，对于单个文件，只适合一个客户端**顺序大批量的写入**。在单个文件上，它的高性能是指高吞吐量。而它所支持的高并发，则是很多个不同的应用，通过不同的客户端去读写不同的文件。这样，它的并发会分配到不同的 chunkserver 上去，互相之间也不会有竞争。

日志收集系统

所以，为了更适合 HDFS（GFS）这样的特性，我们需要一个中间层来帮助解决问题。这个中间层，通常被称为**日志收集器**（Log Collector）。其中，Facebook 推出了开源的 Scribe，Cloudera 推出了 Flume。

这些日志收集系统的架构也并不难懂。简单来说，就是各个应用服务器上，有一个日志收集器的客户端。多个客户端，会把日志发送到一个日志汇集（Log Aggregator）的服务器里。而多个日志汇集的服务器，还可以再次用类似的方式进行汇集。这样，通过一个类似于**多层树状**的结构，最终只有几个日志汇集服务器会向 HDFS 写入数据。

这种方式，使得我们既不会有太多的并发写入请求，直接打到 HDFS 上，同时又尽可能地发挥了，HDFS 顺序写入数据高吞吐量的优势。



Scribe这样的日志收集器，作为应用服务器和HDFS之间的中间层
避免了大量应用服务器的请求都要直接访问HDFS的并发压力问题

而且，这些系统本身，就已经设计了各种容错机制。比如，在网络传输中断的情况下，Scribe 会先把数据写入到本地磁盘，等待网络恢复的时候再做“断点续传”。不过，在 2011 年这个时间节点，像 Scribe 这样的系统，仍然不是一个流式数据处理系统，而只是一个日志收集器（Log Collector）。实际上，它并不是实时不断地向 HDFS 写入数据，而是**定时地向 HDFS 上转存（Dump）文件**。

只不过，这个“定时”很频繁，比如每 5 分钟，甚至每 1 分钟就向 HDFS 上写入一个日志文件。不过，因为我们已经做了所有应用服务器的日志汇集，即使有 1000 台服务器，这一分钟也只有一个日志文件，并不会遇到大量碎片化的小文件的问题。

而能够每分钟都把最新的日志文件放上 HDFS，就使得数据分析的工作也能够按分钟进行了。虽然说，这些分析工作仍然是通过频繁运行 MapReduce 任务来进行的。但是，能够更及时地获取实时的数据反馈，已经对实际的广告效果、搜索质量产生了很大的帮助。

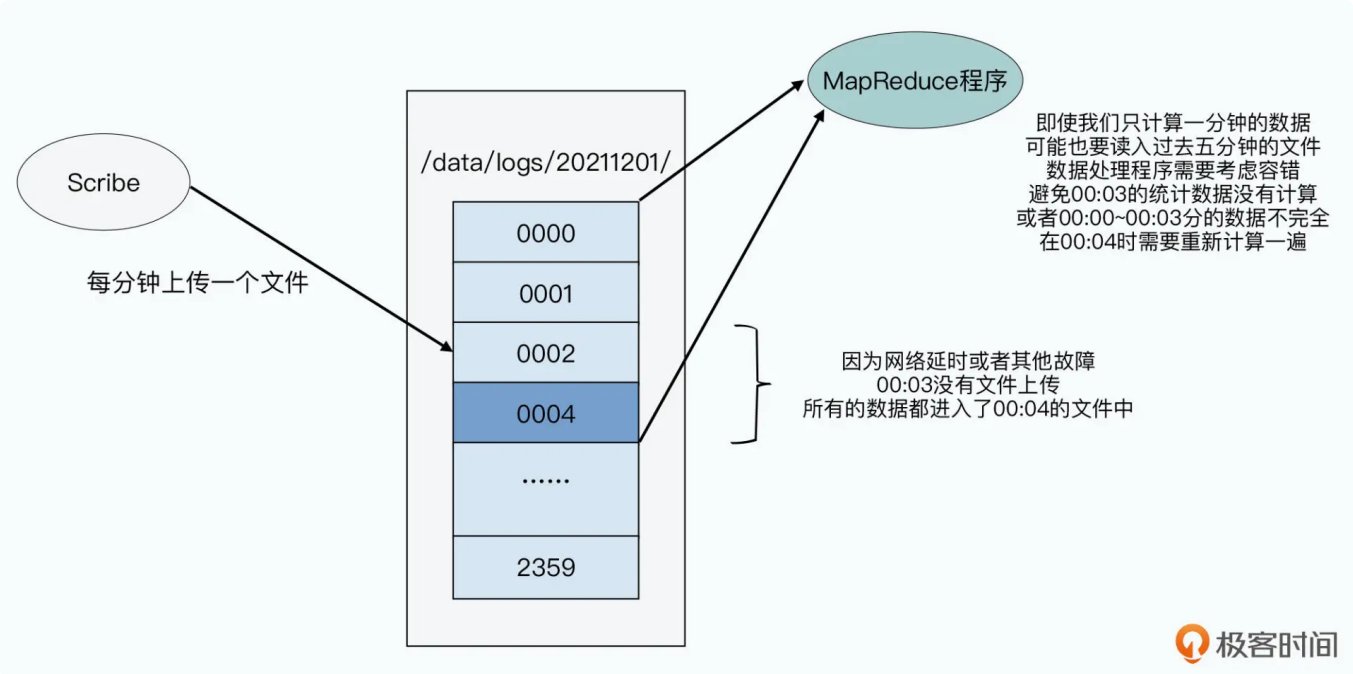
实时数据处理推动 Kafka 的诞生

不过显然，每分钟运行一个 MapReduce 的任务，不是一个高效的解决问题的办法。而且，在这个机制下，日志传输的 Scribe 和进行数据分析的 MapReduce 任务之间，还有很多“**隐式依赖**”，并且使得实际的数据分析程序，需要考虑对于 Scribe 这样的日志传输系统的**“容错”问题**。

比如说，数据分析程序，往往想要分析最近 1 分钟、5 分钟的广告点击的数据。那么，Scribe 就需要每分钟生成一个新文件，放到 HDFS 上。而且，这个文件的文件名需要能够分辨出来，这是哪一分钟的日志。

可是光这样还不够，因为 Scribe 里，也可能会出现网络中断、硬件故障等等的问题，所以我们很有可能，在运行 MapReduce 任务去分析数据的时候，Scribe 还没有把文件放到 HDFS 上。那么，我们的 MapReduce 分析程序，就需要对这种情况进行容错，比如，下一分钟的时候，它需要去读取最近 5 分钟的数据，看看 Scribe 上传的新文件里，是否会有本该在上一分钟里读到的数据。

而这些机制，意味着下游的 MapReduce 任务，需要去了解上游的日志收集器的实现机制。并且两边根据像文件名规则之类的隐式协议产生了依赖，这就使得数据分析程序写起来会很麻烦，维护起来也不轻松。



Scribe和MapReduce通过文件名隐式地传递信息，MapReduce也要考虑Scribe的延时等潜在的不可靠问题

这也是为什么，像 Storm 和 Kafka 这样的系统站上了历史舞台。上节课里，我们已经对 Storm 有所认识和了解了。那么 Kafka，是我们使用 Storm 里必然会使用的一个环节。

Kafka 的系统架构

首先，我们仍然可以把 Kafka 看成是一个类似于 Scribe 这样的日志收集器。上游的应用服务器仍然会把日志发送给 Kafka 集群，但是在 Kafka 的下游，它不仅能把对应的数据，作为文件上传到 HDFS 上。同时，像 Storm 这样的流式数据处理系统，它的 Spout 会直接从 Kafka 里获取数据，而不是从 HDFS 上去读文件。这个时候，Kafka 其实变成了一个**分布式的消息队列**。

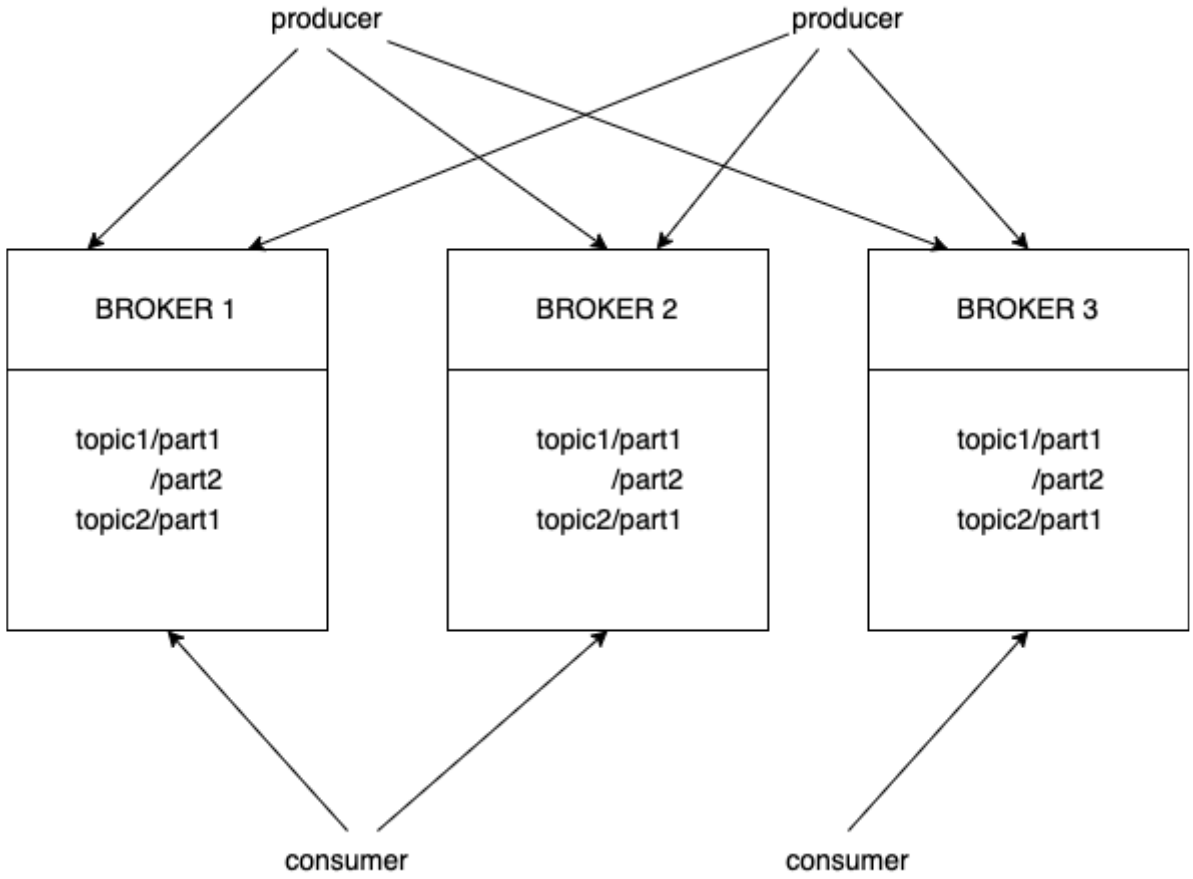
Kafka 的整个系统架构和概念并不复杂，和你日常见过的消息队列一样，它是一个**典型的生产者（Producer）- 消费者（Consumer）模型**。在 Kafka 里，有这样几个角色。

首先是 **Producer**，也就是日志的生产者，通常它就是我们前面的应用服务器。应用服务器会生成日志，作为生产者，把日志发送给到 Kafka 系统中去。

然后是 **Broker**，也就是我们实际 Kafka 的服务进程。因为为了容错和高可用，Kafka 是一个分布式的集群，所以会有很多台物理服务器，每台服务器上都会有对应的 Broker 的进程。Kafka 会对所有的消息，进行两种类型的分组。

第一种，是根据**业务情况**进行分组，在 Kafka 里，对应的就是 Topic（主题）这个概念。比如我们可能既有广告日志，又有搜索日志，两种日志的格式和用途都不一样，那么我们就会通过 Topic 区分开来。

第二种，是进行**数据分区**，这个和我们见过的其他分布式系统进行分区的原因是一样的。一方面，分区可以帮助我们水平扩展系统的处理能力；同一个 Topic 的日志，可以平均分配到多台物理服务器上，确保系统可以并行处理。另一方面，这也是一个有效的“容错”机制，一旦有某一个 Broker 所在的物理服务器出现了硬件故障，那么上游的 Producer，可以把日志发到其他的 Broker 上，来确保系统仍然可以正常运作。



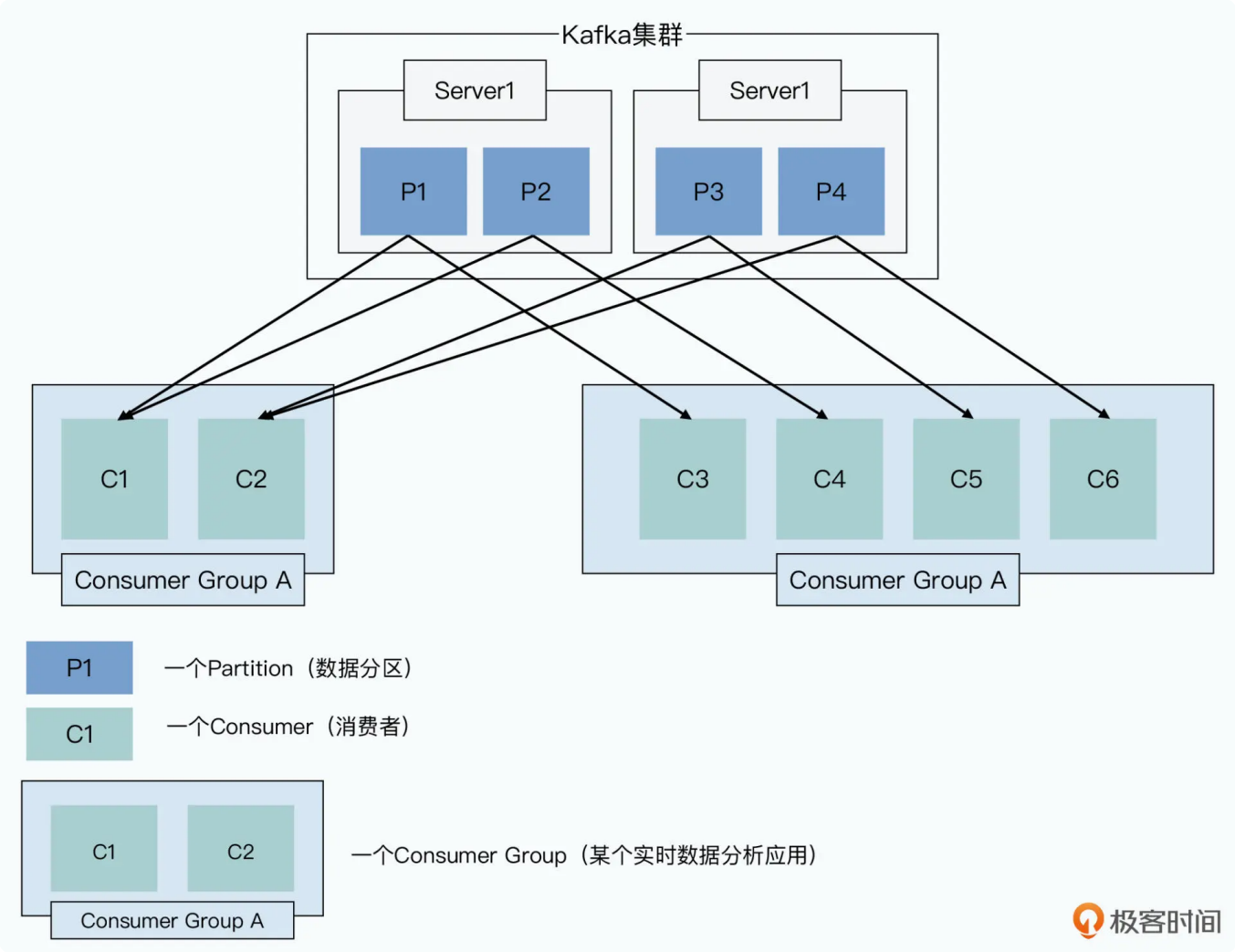
论文中的图1，Kafka本身的架构并不复杂，业务上通过Topic划分日志，水平扩展层面通过Partition进行分区

最后是 **Consumer**，也就是实际去处理日志的消费者。我们去读取 Kafka 数据，把它放到 HDFS 上的程序，就是一个消费者。而我们去获取实时日志，进行分析的程序，也同样是一个消费者，比如一个已经提交运行的 Storm Topology。Kafka 对于它所处理的消息，是支持多个 Consumer 的，这个可以从两个层面来看：

首先，是同样一条消息，可能有不同用途的应用程序都需要读取，它们都是 Kafka 的消费者。比如上传日志到 HDFS 是一个 Consumer，Storm 的 Toplogy 是另一个 Consumer。

其次，是同一个用途的应用程序，可以有多个并行的消费者，来同时并行处理数据，确保下游的 Consumer 有足够的吞吐量。

为了区分这两种“多个消费者”代表的不同含义，Kafka 把每一个用途的 Consumer 程序，称之为一个 **Consumer Group**。也就是说，Kafka 里，会有很多个不同的 Consumer Group，它们会根据自己的用途去消费相同的消息。而一个 Consumer Group 里，会有很多个 Consumer，不同 Consumer 之间分摊压力，会去消费不同的消息。



理解Partition（分区）、Consumer（消费者）、Consumer Group（消费者组）之间的关系

到这里，你可能会觉得，看起来 Kafka 也没有什么特别的呀。似乎和一般的消息队列的功能差不多，我们通过配置一下 Scribe 这样的日志收集器，同样也能够实现类似的功能呀。我们把下游有哪些消费者，写到一个配置文件里。通过读取配置文件，也部署一些 Scribe 进程，把汇集的日志向下游也发起一份就好了。

的确，在遇到 Kafka 之前，我自己就是这样通过 Storm 提供的 ScribeSpout，来进行实时数据处理的。也就是由上游的流式传输系统，主动向下游“推（Push）”数据。

拉数据而不是推数据

但是，这个主动推送数据到下游的方案，其实有一个很严重的缺陷，那就是**消息队列本身，需要维护下游是否已经成功处理消息这个状态。**

你可以回想一下，我们上节课讲解过的 Storm 的“至少一次”的消息处理机制。我们在 Storm 里，是要等整个消息走完 Topology 之后，才能确认消息已经处理完成，这个时候，AckerBolt 会告诉 Spout，这个消息处理完了。但是在这整个过程结束之前，Spout

必须一直把这一条消息保留在内存中。如果这个时候，Spout 进程挂掉了，会发生什么事情呢？

首先，Spout 内存里的消息都消失了。这个时候，如果我们想要让系统真的能够做到对于消息进行“至少一次”的处理，我们就需要上游的 Scribe 向我们重新发送一遍这个消息。那么，在 Scribe 这一端，我们就需要去维护，哪些消息发送到了哪些 Spout，这些消息是否已经处理完成了。并且在消息处理失败，或者超时的时候，重新发送消息给到 Spout。

如果我们的下游，有大量不同的 Consumer Group，我们对每一个 Consumer Group 都要维护这样一份信息，那么就会占用大量的内存。而如果上游不去关心，下游是否真的已经处理完成了数据，那么下游的 Storm 所说的“至少一次”的消息处理机制，就成了一句空话，根本实现不了。

事实上，不只是 Scribe 这样的日志收集器会遇到这个问题，传统的消息队列也会有类似的情况。传统的消息队列，通常会通过一个 message-id 来唯一标识一条消息，只有当下游的所有订阅了这个消息的消费者，处理完成之后，消息队列就认为这条消息被处理完成了，可以从当前的消息队列里面删除掉了。但是，这个机制也就意味着，这个消息队列在下游数据分析完成之前，需要一直存储着这些消息，等待下游的响应，会消耗大量的资源。

而 Kafka 则采用了一个完全不同的方式来设计整个系统，简单来说，就是两点：

第一点，是**让所有的 Consumer 来“拉取”数据**，而不是主动“推送”数据给到 Consumer。并且，Consumer 到底消费完了哪些数据，是由 Consumer 自己维护的，而不是由 Kafka 这个消息队列来进行维护。

第二点，是采用了一个非常简单的**追加文件写的方式**来直接作为我们的消息队列。在 Kafka 里，每一条消息并没有通过一个唯一的 message-id，来标识或者维护。整个消息队列也没有维护什么复杂的内存里的数据结构。下游的消费者，只需要维护一个此时它处理到的日志，在这个日志文件中的偏移量（offset）就好了。

然后，基于这两个设计思路，Kafka 做了一些简单的限制，那就是一个 consumer 总是顺序地去消费，来自一个**特定分区**（Partition）的消息。而一个 Partition 则是 Kafka 里面可以并行处理的最小单位，这就是说，一个 Partition 的数据，只会被一个 consumer 处理。

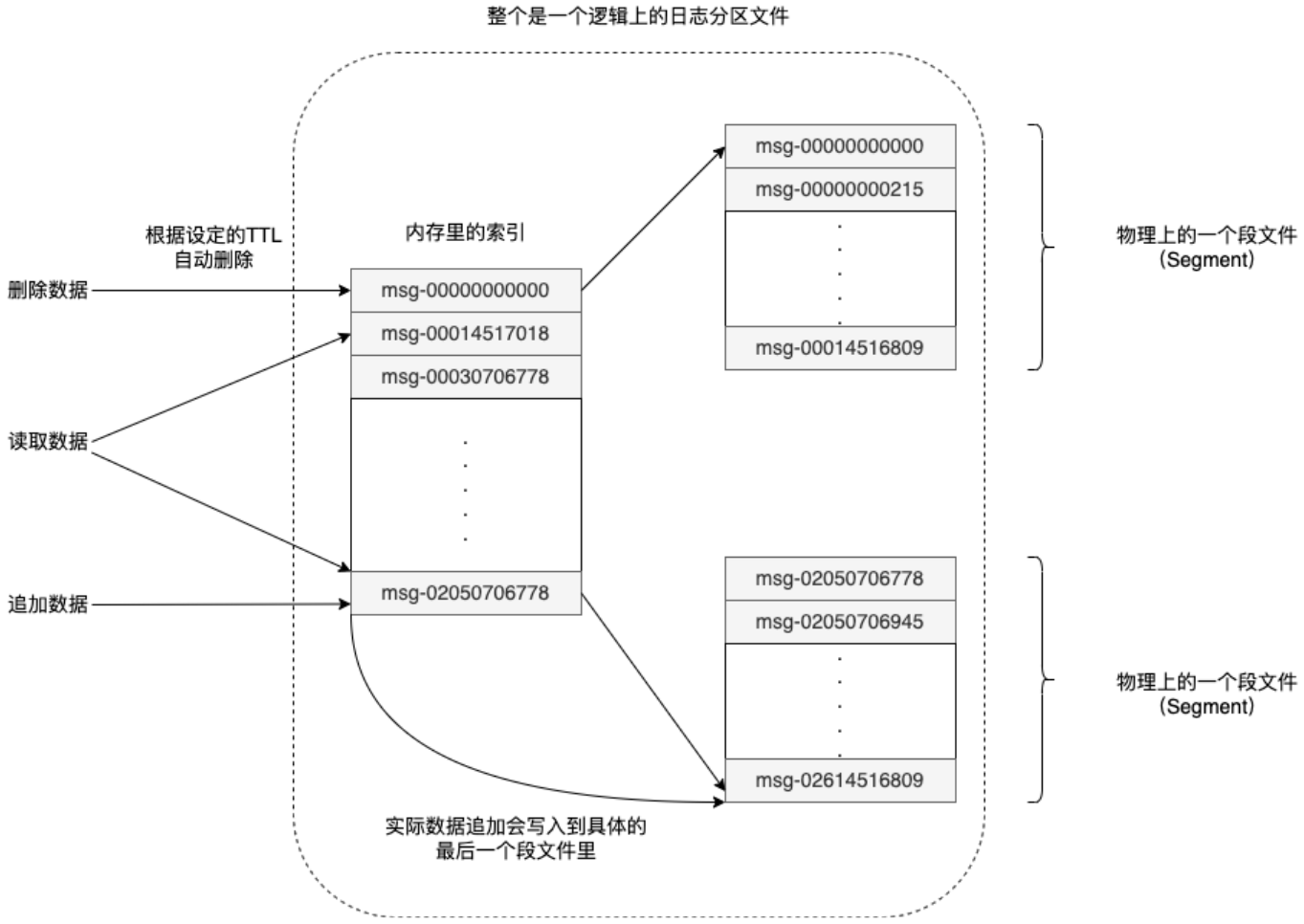
这样一来，整个 Kafka 的系统设计也一下子变得特别简单。所有的 Producer 生成消息，和 Consumer 消费消息，都变成了简单的顺序的文件读和文件写。而我们知道，硬盘的顺序读写的性能要远高于随机读写。

Kafka 的单个 Partition 的读写实现

在实际的实现上，Kafka 是这么做的。每一个 Topic 会有很多个 Partition，分布到不同的物理机器上。一个物理机上，可能会分配到多个 Partition。实际存储的时候，我们的一个 Partition 是一个逻辑上的日志文件。在物理上，这个日志文件会给实现成一组大小基本相同的 Segment 文件，比如每个 Segment 是 1GB 大小。每当有新消息从 Producer 发过来的时候，Broker 就会把消息追加写入到最后那个 Segment 文件里。

而为了性能考虑，Kafka 支持我们自己设置，是每次写入到把数据刷新到硬盘里，还是在写入了一定数量的日志或者经过一个固定的时间的时候，才把文件刷新到硬盘里。

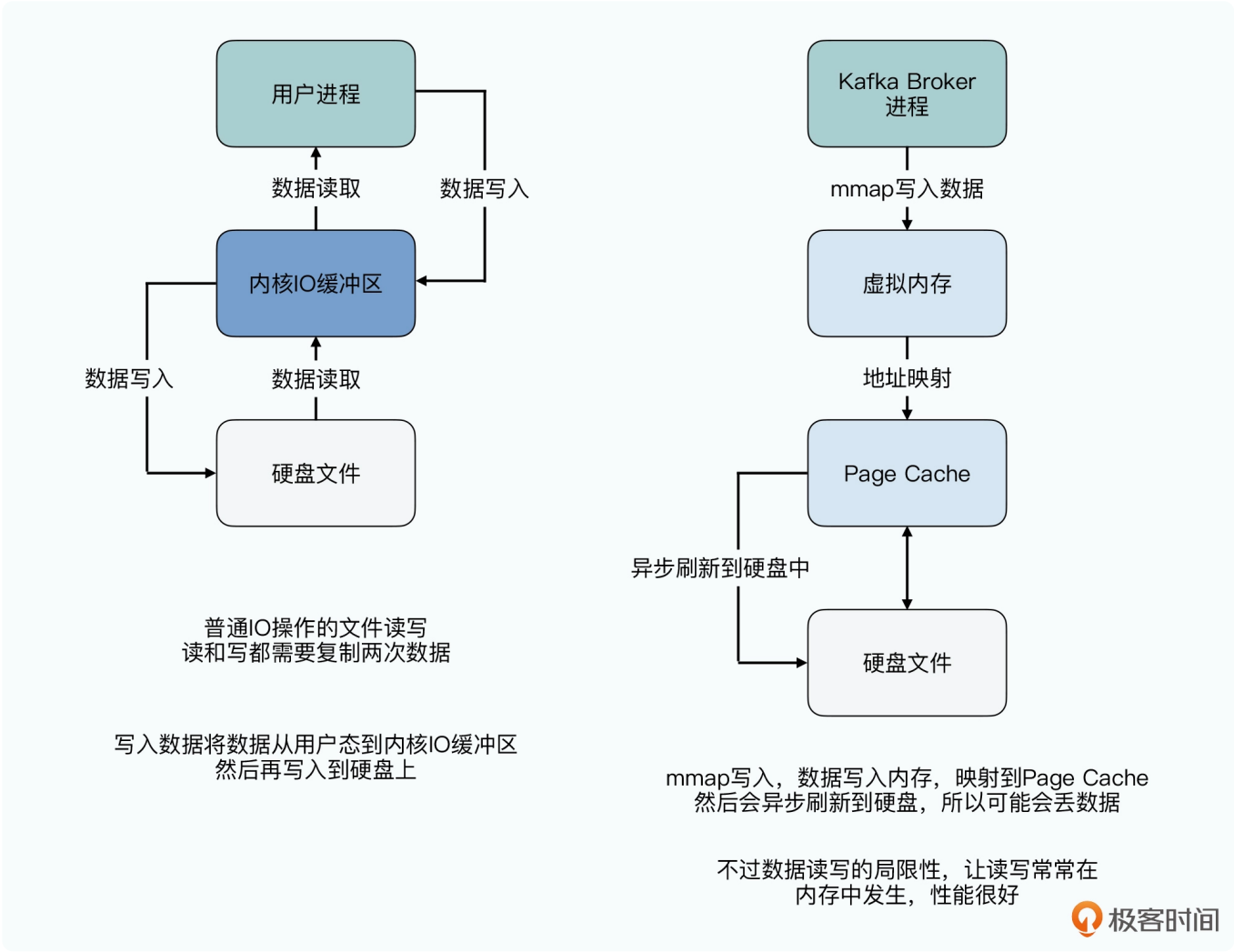
Broker 会在内存里维护一个简单的索引，这个索引其实就是每个通过一个虚拟的偏移量，指向一个具体的 Segment 文件。那么在 Consumer 要消费数据的时候，就是根据 Consumer 本地维护的已经处理完的偏移量，在索引里找到实际的 Segment 文件，然后去读取数据就好了。



论文中的图2，Partition是一个逻辑上的日志文件，在物理上会拆分成多个同样大小的Segment文件，顺序存储日志

优秀的 Linux 文件系统

因为本质上，Kafka 是直接使用本地的文件系统承担了消息队列持久化的功能，所以 Kafka 干脆没有实现任何缓存机制，而是**直接依赖了 Linux 文件系统里的页缓存**（Page Cache）。Kafka 写入的数据，本质上都还是在 Page Cache。而且因为我们是进行流式数据处理，读写的数据有很强的时间局部性，Broker 刚刚写入的数据，几乎立刻会被下游的 Consumer 读取访问，所以大量的数据读写都会命中缓存。



Kafka直接通过mmap（Memory Mapped Files）来写入数据，使得数据读写的性能很好

而没有自己在内存里面实现缓存，也避免了两个问题。

第一个是 JVM 里面的 GC（垃圾回收）的开销。如果我们有大量的消息是缓存在内存里，那么处理完了之后，就需要通过 GC 销毁这些对象，腾出空间来容纳新的需要缓存的对象，而 JVM 的 GC 开销，可能会短时间大幅度影响 Broker 的性能。

第二个是缓存的“冷启动问题”。如果我们的 Broker 进程挂掉了，重新启动了一个新的进程，那么此时，我们的内存里是没有任何缓存数据的，这个时候读取数据的性能，会比一个已经长时间运行、内存中缓存了很多数据的系统的性能，差上很多。

这两点，都会导致系统本身的性能抖动。而通过直接利用文件系统本身的 Page Cache，我们的 JVM 内除了基本的业务逻辑代码，没有其他的内存占用和 GC 开销。

除了利用文件系统之外，Kafka 还利用了 Linux 下的 **sendfile API**，通过 DMA 直接将数据从文件系统传输到网络通道，所以它的**网络数据传输开销也很小**。关于这个主题，我在

《深入浅出计算机组成原理》的《DMA：为什么 Kafka 这么快？》这节课专门讲解过，你也可以去看一下。

小结

好了，到这里呢，我们已经把 Kafka 的整体系统架构，以及单个 Partition 上的数据生产和消费讲解完了。其实，**Kafka 之所以在大数据领域，比 Scribe 这样的日志收集系统，以及传统的消息队列要好用的原因，在于这些系统对于业务需求的假设是不同的。**

Kafka 的假设是，我们处理的是互联网领域的海量日志，我们对于丢失一部分日志是可以容忍的。因为几 TB 的广告浏览和点击日志少了几条，其实并不会对业务产生什么影响。但是，我们**需要关注系统整体的吞吐量、可扩展性、以及错误恢复能力**。

而传统的消息队列，则关注的是小数据量下，是否每一条消息都被业务系统处理完成了。因为这些消息队列里的消息，可能就是一笔实际的业务交易，我们需要等待 consumer 处理完成，确认结果才行。但是整个系统的吞吐量却没有多大。

而像 Scribe 这样的日志收集系统，考虑的是能否高吞吐量地去传输日志，至于下游如何处理日志，它是不关心的。

而 Kafka 的整体设计，则主要考虑的是我们**不仅要实时传输数据，而要开始实时处理数据**了。我们需要下游有大量不同的业务应用，去消费实时产生的日志文件。并且，这些数据处理也不是非常重要的金融交易，而是对于大量数据的实时分析，然后反馈到线上。

而且，下游消费数据的，可能有很多个不同的团队、业务、产品。所以，在设计上，Kafka 采用了让 Consumer 自己拉数据并且维护数据处理的进度，把下游的业务处理和上游的数据流式传输解耦开来了。而通过利用 Linux 文件系统和硬盘的高性能顺序读写的硬件特性，Kafka 实现了非常高的吞吐量，最大程度上匹配了“大数据”这个主题。

那么在下节课里，我们会继续深入来看一下，Kafka 的“分布式”部分又是如何搭建的。以及我们之前说过的 Lambda 数据处理架构，和基于 Kafka 的 Kappa 处理架构又是怎么一回事儿。

推荐阅读

其实，Kafka 本身的设计并不复杂。不过，**Kafka 给我们带来的一个重要的思考是，我们不能简单地只从一个系统内部来思考它应该怎么设计，而是要考虑全链路的数据流程有哪些需求。**

所以，我推荐你去读一读《Realtime Data Processing at Facebook》这篇论文。它可以帮助你从全局视角，从应用层面看大数据系统的整体设计是怎么样的。从这个视角看问题，会让你在设计系统的时候，不只考虑系统内部的架构、性能，也要考虑外部的其他人，会如何使用你的系统。而这一点，在大型团队和大型系统中是非常重要的一个环节。

思考题

我们今天在讲解 Kafka 的时候，其实没有讲过 Kafka 是如何做到“高可用的”，在论文的原文里面，也没有深入讲解 Kafka 是如何容错、以及保持高可用的。那么，通过你对课程之前内容的学习，以及这节课了解的 Kafka 的整体架构，你觉得 Kafka 可以怎么做到高可用呢？

欢迎在留言区分享你的思考和答案，和其他同学一起交流，共同进步。谢谢收听，咱们下节课再见。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 26 | 从S4到Storm（二）：位运算是个好东西

下一篇 28 | Kafka（二）：从Lambda到Kappa，流批一体计算的起源

精选留言 (3)

写留言

在路上
2021-12-08

《Realtime Data Processing at Facebook》这篇论文帮助我建立了对流式处理系统的认识。业务系统允许秒级延迟，而不是毫秒级延迟，使得系统之间可以通过Kafka连接。使用Kafka传输数据带来了额外的好处：
1. 容错：流式处理节点的故障变得独立。
2. 容错：恢复故障变得更快，只需要在其他地方启动一个相同的节点。...

展开

在路上
2021-12-08

徐老师好，读完2011年的Kakfa论文，我惊讶的发现那时候的分区数据居然没有副本，充分体现了Kafka对业务需求的假设，可以容忍丢失部分日志。Kafka在未来工作中提到，第一、会增加同步和异步的副本模式，并由用户根据业务场景来选择需要的副本模式，第二、会增加流式处理的能力。从最新的Kafka版本来看，Kafka已经实现了这个目标，其中灵活的副本模式就是ISR机制。...

展开

101 酥

2021-12-06

备份机制：同一个 partition 存在多个数据副本：leader & follower
ISR 机制：在指定容错时间内，与 leader 保持数据同步的副本机制
ACK 机制：生产者发送消息后，消费者收到消息后的确认机制
故障恢复机制：leader 选举及失败恢复

展开